(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 014 263 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
28.06.2000 Bulletin 2000/26

(51) Int Cl.[7]: G06F 9/445

(21) Application number: 99310012.2

(22) Date of filing: 13.12.1999

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 14.12.1998 US 212182

(71) Applicant: APPLIED MICROSYSTEMS CORPORATION
Redmond, Washington 98072-9702 (US)

(72) Inventor: Tinker, Jeffrey L.
Kenmore, Washington 98027 (US)

(74) Representative: Driver, Virginia Rozanne et al
Page White & Farrer
54 Doughty Street
London WC1N 2LS (GB)

(54) Method and system for modifying executable code to add additional functionality

(57) A system for modifying a compiled executable code file by adding patches that add functionality when the modified executable code file is executed. The modifying is performed without recompiling, relinking or rewriting the code file. Adding a patch involves creating a patch handler which when executed causes the patch statements to be executed, and may involve replacing one or more existing compiled instructions in the file with patching instructions to transfer flow of execution to the appropriate patch handler. The instructions replaced by the patching instructions can also be added to the patch handler. Patches can also include code statements which form a complete module, such as an invokable routine, which can be referenced by other patch statements to cause the module to be executed. Specialized trace requests can also be added as patch statements. The trace requests will make specified information about the current execution of the executable code file available to a software developer. Patch statements can include variables and expressions that will be evaluated in the context of the appropriate current variable scope, regardless of whether the scope is defined within the patch or by existing compiled instructions. After patches have been added, they can be disabled so as to prevent their execution without removing the patching instructions from the compiled executable file. Patches can also be qualified with conditions such that the patch will be executed only when the conditions are true at the time of execution.

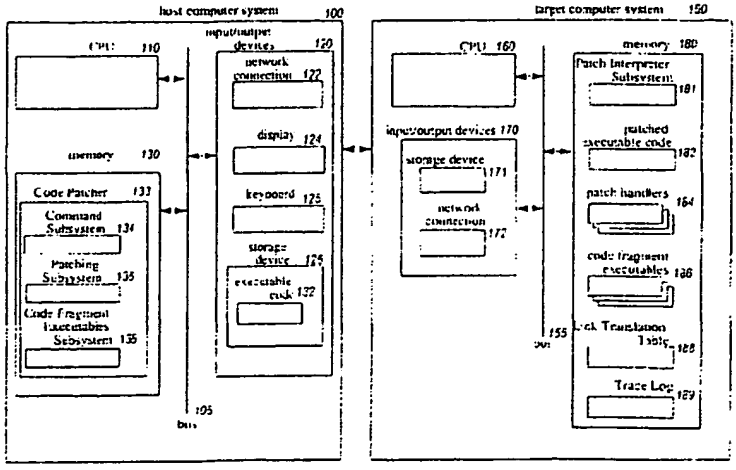


Fig. 1

EP 1 014 263 A2

Description

[0001] The present invention relates generally to efficiently creating executable software, and more particularly to modifying compiled executable files to add additional functionality.

[0002] In the past, creating executable software code could be a time-consuming task. The typical code creation process involved first creating a source code program (*i.e.*, a series of lines of program statements in a high-level computer language) with a text processing program, compiling and linking the source code (sometimes with an intermediate assembling step) to create executable code (*i.e.*, a series of machine language instructions) for a specified computer processor, and then storing the executable code in an executable file. The executable code could then be debugged by executing the executable file on the specified computer processor to determine if the software performed its task correctly, or if instead it had one or more errors (*i.e.*, bugs). If the executable code had errors, a software developer would modify the source code in an attempt to remove the errors, recompile the source code, and then link the recompiled code to produce a new executable file for debugging. For large software programs, this process was iteratively performed a large number of times until all known errors were removed.

[0003] In many cases, the cause of an error (*e.g.*, a mistake in the program logic) is not obvious from executing the executable file. Various options existed for a software developer to identify errors. For example, a software developer could add print statements throughout the source code so that as the corresponding executable print instructions are executed (after compiling and linking), they would report the current progress of the execution. Knowledge of the current execution progress assists in identifying the section of the code which was executing when an error occurred. In addition to merely reporting execution progress, print statements can also display the current value of variables or source code expressions at specified points throughout the execution. Since the print statements were part of the original compilation/linking process, the variables and expressions that were part of the print statements would be evaluated in the context of the current variable scope (*e.g.*, using the value of a local variable in a currently executing function rather than a variable with the same name in a different non-executing function), as would any other compiled code statement.

[0004] In addition to print requests, application programs known as debuggers were developed to provide additional control over execution of executable files. A debugger loads executable code into memory and then controls execution of the executable code. For example, the debugger can execute a single executable code instruction at a time. Alternately, the debugger can execute the executable code continuously until a break point designated within the debugger is reached. Such debuggers can also use additional information stored in an executable code file during the compiling and linking steps to reconstruct and display the source code lines that correspond to the instructions in the executable code. The display of the source code facilitates control by the software developer of the execution of the executable code (*e.g.*, setting a breakpoint at a particular point in the source code). When execution of the executable code is stopped, a user can interact with the debugger to view current values of variables and expressions. In addition, some debuggers allow a user to view the effects of temporarily modifying a source code line. Nonetheless, although such debuggers can assist with locating errors in executable compiled code, recompiling and linking is needed to fix errors that are located.

[0005] In addition to the use of debuggers, other techniques have been developed to modify the functionality of compiled code without requiring a full recompilation and linking. One technique involves relinking previously compiled code with different code than was previously used for linking (*e.g.*, using an updated Dynamic-Link Library or replacing a stubbed routine with a functional routine). In this situation, no changes are made to the previously compiled code, but changes in the overall program functionality can occur due to the different operation of the newly linked code. However, this technique is not typically useful in modifying errors in the compiled code (since the compiled code is not changed) or in flexibly adding functionality to an executable file at a desired user-specified location (since only previously specified link points can be used for the relinking).

[0006] Another technique to modify the functionality of compiled code without requiring recompilation and linking involves rewriting an executable file. Rather than modifying an existing file, rewriting involves creating an entirely new executable compiled file based on an existing executable file. Rewriting an executable file does allow new functionality to be added to an executable file at a user-specified location because new compiled instructions can be added to the new file. However, rewriting is difficult to perform without adding errors into the new file, and the specific mechanisms for adding instructions (*e.g.*, adjusting offsets in existing instructions) typically vary on each type of processor.

[0007] When executable code is being created for an embedded system (*e.g.*, an embedded controller for manufacturing equipment), the problems with software code creation are exacerbated. Such embedded systems may include only a CPU and memory, without having access to other standard computer system components such as a keyboard or display. In addition, standard application programs such as text processors and debuggers may not be available for an embedded system. In this environment, the source code will typically be created on a host computer system separate from the embedded target system. This allows a user application such as a text processor to create the source code. The source code is then compiled for the target

embedded computer system and transferred (*e.g.*, over a network) to the embedded system for execution and debugging. When an error occurs during execution of the executable code on the embedded system, the lack of standard computer system components and application programs on the target system make it extremely difficult to determine the cause of the error. Even obtaining information about the current state of the execution at the time of the error is typically difficult. Moreover, even if such information is available, it will need to be transferred back to the host computer system where modifications to the source code can begin another compile/link/transfer/debug cycle.

[0008] In accordance with one aspect of the present invention, a method and system are provided for modifying a compiled executable code file so as to add functionality when the modified executable code file is executed. The modifying of the executable code file is performed without recompiling, relinking or rewriting the executable code file. In particular, the system allows a user to indicate changes to be made to the compiled executable file, including adding code statements to the compiled executable file and manipulating previously added code statements in a variety of ways. Each set of code statements which are to be added is referred to as a patch, with each statement in a patch being referred to as a patch statement. After the patches have been specified, the system modifies the compiled executable code in a non-transitory manner such that the patch statements will be performed when the modified executable code is executed in the future. This manner of modification allows the traditional compile/link/debug cycle to be avoided, thus providing significant time savings.

[0009] Adding a patch to a compiled executable code file involves creating a patch handler which when executed causes the patch statements to be executed and which may additionally perform various housekeeping functions. The patch statements may be stored in an executable machine-independent non-compiled format, and if so are interpreted when executed on a target computer. Adding a patch may also involve replacing one or more existing compiled instructions in the compiled executable code file with patching instructions which, when executed, will transfer flow of execution to the appropriate patch handler loaded in memory. When a patch is intended to be executed in addition to the existing compiled instructions (rather than substituting for one or more instructions), the existing instructions replaced by the patching instruction can be added to the patch handler so that they will be executed in addition to the patch statements. The patching instructions can directly identify the memory location of a patch handler, such as with a transfer instruction. Alternately, the patching instructions can include a unique reference to the appropriate patch handler, and if so the appropriate memory location for the referenced patch handler will not be determined until the time of execution. Patches can also include code statements which form a complete module, such as an invokable routine, which can be referenced by other patch statements to cause the module to be executed. In addition, patch statements can include variables and expressions that will be evaluated in the context of the appropriate current variable scope, regardless of whether the scope is defined within the patch or by existing compiled instructions. Finally, specialized trace requests can also be added to the compiled executable code as patch statements. The trace requests will make specified information about the current execution of the executable code file available to a software developer, such as by storing it on a local trace log file or transmitting it electronically. After patches have been added, they can be disabled so as to prevent their execution without removing the patching instructions from the compiled executable file. Patches can also be qualified with conditions such that the patch will be executed only when the conditions are true at the time of execution.

[0010] In accordance with one aspect of the present invention, the system modifies on a target computer a compiled file executable on the target computer, with the compiled file including a plurality of compiled instructions. The modifying is performed under control of a source computer, and it involves first loading the compiled file onto the target computer and then receiving an indication to modify the compiled file by adding at least one instruction to be executed upon execution of the compiled file. The system then creates a patch group having instructions including the indicated instructions. The system then modifies the compiled file on the target computer by replacing an instruction in the compiled file with a patch instruction, and loads the patch group into a portion of memory on the target computer distinct from the memory in which the compiled file is loaded. The system then executes on the target computer the instructions in the modified compiled file by, when an instruction to be executed is the patch instruction, indicating one of the plurality of instructions in the loaded patch group as a next instruction to be executed.

[0011] For a better understanding of the present invention and to show how the same may be carried into effect, reference will now be made by way of example to the accompanying drawings, in which:-

[0012] Figure 1 is a block diagram illustrating an embodiment of the Code Patcher system of the present invention.

[0013] Figures 2A and 2B illustrate an example of patched software code.

[0014] Figures 3A and 3B illustrate examples of source code in an executable computer-independent non-compiled parse tree format.

[0015] Figure 4 is an exemplary flow diagram of an embodiment of the Command Subsystem routine.

[0016] Figure 5 is an exemplary flow diagram of an embodiment of the Process Patch Request subroutine.

[0017] Figure 6 is an exemplary flow diagram of an embodiment of the Patching Subsystem routine.

[0018] Figure 7 is an exemplary flow diagram of an embodiment of the Code Fragment Executables System routine.

[0019] Figure 8 is an exemplary flow diagram of an embodiment of the Patch Interpreter Subsystem routine.

[0020] Figure 9 is an exemplary flow diagram of an embodiment of the Execute Patched Executable Code subroutine.

[0021] An embodiment of the present invention provides a method and system for modifying a compiled executable code file so as to add functionality when the modified executable code file is executed. The modifying of the executable code file is performed without recompiling, relinking or rewriting the executable code file. In particular, the Code Patcher system loads compiled executable code into memory on a target computer, and allows a user to indicate patches to be made to the compiled executable file. These patches can include adding code statements (*i.e.*, compiled instructions or lines of source code) to the compiled executable file. In one embodiment, the source code lines corresponding to the compiled executable instructions in the file can be shown, and the user can indicate the changes to be made in the displayed source code. After the patches have been indicated, the Code Patcher system modifies the compiled executable code in a non-transitory manner such that the patches will be performed when the modified executable code is executed in the future. This manner of modification allows the traditional compile/link/debug cycle to be avoided, thus providing significant time savings.

[0022] Figure 1 illustrates an embodiment of a Code Patcher system 133 in which the patching of a compiled executable code file occurs on a host computer system 100 that is separate from a target computer system 150 on which the patched executable code is to be executed. Compiled executable code files to be patched contain compiled instructions which are executable as native code on the CPU of target computer system 150, and which may or may not be executable as native code on the CPU of host computer system 100. In addition, in the illustrated embodiment the target computer system 150 has a RISC processor in which all instructions are the same length, so one compiled instruction will require the same memory space as another compiled instruction. Moreover, code statements are converted into a machine-independent non-compiled format that will be executed interpretively on the target computer system 150. Those skilled in the art will appreciate that code statements could alternatively be compiled and that the Code Patcher system 133 is not limited to patching code for RISC processors.

[0023] The host computer system 100 includes a CPU 110, input/output devices 120, a memory 130, and a bus 105. The input/output devices include a storage device 126, a network connection 122, a display 124, and a keyboard 125. The Code Patcher system 133 is executing in memory 130, and can be used to patch compiled executable code files such as compiled executable code file 132 stored on the storage device 126. The Code Patcher system 133 can also be stored on a storage device (not shown) such as storage device 126 before being loaded into memory 130, and the compiled executable code file 132 may also be loaded into memory 130 to assist operation of the Code Patcher system (*e.g.*, for display to a user).

[0024] The target computer system 150 similarly includes a CPU 160, input/output devices 170, the memory 180, and a bus 155. The input/output devices 170 include a storage device 171 and a network connection 172. A Patch Interpreter Subsystem 181 is executing in memory 180 to assist in the execution of one or more patched executable code files on target computer system 150. The executable code file 132 is loaded into memory 180 when it is to be executed. After the executable code file is patched and becomes patched executable code file 182, the patch handlers 184 and code fragment executables 186 that are associated with the patched executable code file are also loaded into memory 180. The memory 180 also includes some or all of a Link Translation Table 188 in which current memory locations associated with various identifiers are stored, and of a Trace Log 189 in which information will be stored by the Patch Interpreter Subsystem 181 for later retrieval. In an alternate embodiment, Link Translation Table 188 and Trace Log 189 could be stored on storage device 171 and be accessed by Patch Interpreter Subsystem 181 only when needed. The Patch Interpreter Subsystem 181, patched executable code file 182, patch handlers 184, and code fragment executables 186 can also be stored on a storage device (not shown) such as storage device 171 before being loaded into memory 180.

[0025] When the Code Patcher system 133 is first invoked, the Command Subsystem 134 provides a user interface to a user (not shown) of the host computer system 100. The Command Subsystem 134 allows the user to specify a compiled executable code file that is to be patched, such as code file 132. The Command Subsystem 134 then loads information from executable code file 132 (*e.g.*, from the symbol table) to allow a correspondence to be established between the compiled executable code instructions in the file and the original source code lines (not shown) from which the compiled executable code was created. The Command Subsystem 134 then displays the source code to the user and allows the user to specify a variety of commands.

[0026] After specifying compiled executable file 132, a user can use the Command Subsystem 134 to modify the file by specifying patches. Each set of code statements which are to be added is referred to as a patch, with each added code statement in a patch referred to as a patch statement. Executable code which has been patched (*i.e.*, modified by having a patch added) is similarly referred to as patched code. As is explained in

greater detail below adding a patch to a compiled executable code file involves first creating an executable patch handler which causes the patch statements to be executed (and which may additionally perform various housekeeping functions) and then replacing one or more existing compiled instructions in the file with a patching instruction to transfer flow of execution to the patch handler.

[0027] In the illustrated embodiment, patch statements are stored in an executable machine-independent non-compiled format, and are interpreted (*i.e.*, interpretively evaluated) by an evaluator when executed on target computer system 150 Patch Interpreter Subsystem 181 acts as an evaluator in the illustrated embodiment. Evaluators can be efficiently provided to a variety of target computer systems by cross-compiling an existing evaluator for each of the target systems. In an alternate embodiment, patch statements are stored as the original lines of source code and a source code interpreter (not shown) on target computer system 150 is used to interpret the patch statements. In yet another embodiment, patch statements are compiled for target computer system 150 before execution and are then executed natively on the target computer system.

[0028] A specified patch can take a variety of forms, such as substituting one or more new code statements for one or more existing compiled instructions or lines of source code. After they are substituted for, existing compiled instructions will no longer be executed. Rather than substituting new code statements, a specified patch can also be added such that it is executed in addition to the existing compiled instructions, including even the instructions replaced by patching instructions. This is accomplished by adding the replaced instructions to the patch handler for the patch. Multiple lines or instructions can be included in a patch, even when substituting for a single instruction or when no instructions are being substituted for. Some patches include code statements which form a complete module, such as an invokable routine. If such a module is being added so that it can be invoked by other code statements, rather than to be directly executed at a designated location, it is not necessary to use patching instructions to execute the module. Instead, other patch statements can reference the module and thus transfer flow of execution to the module.

[0029] Other patches include one or more code statements which do not form a complete module, referred to as a code fragment. Such patches will typically require that a patching instruction be added so that flow of execution reaches the code fragment. However, if a patch handler including a code fragment is referenced by another patch handler, the code fragment may not directly require a patching instruction for it to be executed. In addition, patch statements can include variables and expressions that will be evaluated in the context of the appropriate current variable scope, regardless of whether the scope is defined within the patch or by existing compiled instructions. Access to various information for the compiled executable code file to be patched, such as the symbol table, may be needed to resolve references to variables.

[0030] In addition to adding code statements, specialized trace requests can also be added to a compiled executable code file as patch statements. For some target computer systems, such as embedded systems without a display, it can be difficult to gather information about execution of executable code since traditional output mechanisms (*e.g.*, print statements) are unavailable or are difficult to use. The trace requests will store specified information about the current execution of the executable code file in a manner that is accessible to a software developer. For example, if the executable code file is being executed on an embedded computer system, the trace mechanism can store execution information in a trace log file on the embedded computer system for later retrieval, or can transmit the information to a separate computer system for access there. The specified information for a trace request can also include variables and expressions that will be evaluated in the context of the current variable scope at the time of their execution. In addition to reporting the current values of variables and expressions, the trace requests can be used to track each time that a particular instruction or block of code is executed. For example, the first instruction in an invokable procedure can be patched with a trace request indicating the current values of the procedure parameters.

[0031] When a patch is to be added to compiled executable code file 132, the user indicates the patch statements for the patch as well as where the patch is to be added (*e.g.*, in a particular function or at a specific source code or compiled instruction location). In the illustrated embodiment, a patching instruction need replace only a single existing instruction since all RISC instructions are the same size. The user also specifies whether the patch is to substitute for existing compiled instructions or is to be added in addition to the existing instructions. In addition, when a patch is being added in addition to existing instructions, it is determined or specified whether the patch is to be added before or after the existing instructions. When a function is indicated as the patch location, the patch is added at the first instruction for the function (*i.e.*, the patching instruction replaces the first instruction for the function). When a source code line is indicated, the Command Subsystem 134 identifies the one or more corresponding compiled instructions. If multiple instructions correspond to a source code line, one or more of the compiled instructions are selected to be replaced by the patching instruction. The determination of whether the patch is to be added before or after the existing instructions may affect which of the multiple corresponding instructions are selected (*e.g.*, if the patch is added before the instructions, selecting the first of the multiple corresponding instructions).

[0032] The Command Subsystem 134 also allows the

user to perform actions on existing patches, such as to group patches together and to manipulate them either individually or as a group. For example, a patch previously added to the compiled executable code file can be disabled so as to prevent its execution. Patches can also be qualified with conditions such that the patch will be executed only when the conditions are true at the time of execution. In addition, different patches can be grouped together and each group can be manipulated in the same manner as a single patch. For example, all patches in a group can be enabled or disabled, or can be saved for later use. This ability to group, disable/enable, and qualify patches without recompiling the file allows trace requests to be turned on or off in a simple and efficient manner, greatly simplifying debugging. Finally, a user can also use the Command Subsystem 134 to retrieve and view previously created trace information, such as from a trace log.

[0033] After all the patches have been specified and the user is ready to execute the code file 132 as patched, the Command Subsystem 134 notifies a Patching Subsystem 136 of the patches and notifies a Code Fragment Executables Subsystem 138 of the code statements from the patches. The Code Fragment Executables Subsystem 138 converts each sequence of one or more code statements it receives into a code fragment executable 186 in an executable machine-independent non-compiled format. Code fragment executables can be executed in an interpreted manner on any target system with an evaluator for the format. In one embodiment, the code fragment executables are stored as parse trees, as is illustrated further with respect to Figures 3A and 3B. Each code fragment executable will be associated with the corresponding code file being patched, and will also have a unique identifier by which it can be referenced. For example, other statements can use the unique identifier for a code fragment executable representing a module to invoke the module.

[0034] After creating the code fragment executables 186 for the code statements in the patches, the Code Fragment Executables Subsystem 138 then transfers the created code fragment executables to target computer system 150 and updates Link Translation Table 188. Link Translation Table 188 includes a reference for each symbol in each of the code fragment executables 186 which can be referenced by statements outside the code fragment executable (*e.g.*, the unique identifier for a code fragment executable and any exported symbols that can be referenced outside the code fragment executable). Those skilled in the art will appreciate that in an alternate embodiment, Link Translation Table 188 could be stored in an alternate location such as on the host computer system 100. In addition, those skilled in the art will appreciate that the code fragment executables 186 could initially be stored on storage device 126 before being transferred, and could be transferred directly to memory 180 or first to storage device 171.

[0035] When the Patching Subsystem 136 receives notification of the patches from the Command Subsystem 134, the Patching Subsystem 136 creates the appropriate patch handlers 184 and places patching instructions in the appropriate locations in the compiled executable file 182. In addition, in some embodiments, the Patching Subsystem 136 can allocate and deallocate computer system resources on the target computer system 150 needed for the patch handlers 184 (*e.g.*, memory in which to load the patch handlers). The Patching Subsystem 136 first transfers the compiled executable code file 132 to the target computer system 150. After creating the patch handlers 184, the Patching Subsystem 136 transfers the patch handlers 184 to the target computer system 150, and updates Link Translation Table 188 as necessary to include references for the patch handlers 184. The Patching Subsystem 136 also modifies the compiled executable code file on the target computer to include the patching instructions, thus creating patched compiled executable code file 182. Those skilled in the art will appreciate that the patch handlers 184 could initially be stored on storage device 126 before being transferred, and could be transferred directly to memory 180 or first to storage device 171.

[0036] In particular, at each location in the executable code file 132 where a patch is to be added, the Patching Subsystem 136 replaces an existing compiled instruction with a patching instruction, thus removing the replaced instruction from patched compiled executable file 182. Those skilled in the art will appreciate that in an alternate embodiment, an already-patched code file such as file 182 could have additional patches added. In that embodiment, an old patching instruction could be selected as the instruction to be replaced with a new patching instruction. If the new patch is intended to supplement rather than substitute for the previous patch, the replaced old patching instruction can be added to the new patch handler so that the old patch handler will be also executed. Each patching instruction alters the flow of execution to an appropriate patch handler so that the patch statements for the patch will be executed.

[0037] In one embodiment, each path has a distinct patch handler, while in an alternate embodiment a single patch handler can handle all patches. Thus, when the flow of execution reaches a patching instruction, execution will be transferred to and will continue at the appropriate patch handler. In some embodiments, the patching instructions are transfer statements (*e.g.*, jump, goto, branch, call, etc.) which specify an explicit memory location (either directly or via an offset) in which the patch handler will be executed. In alternate embodiments, the patching instructions include a unique reference to the appropriate patch handler, and the appropriate memory location for the referenced patch handler is determined from Link Translation Table 188 at the time of execution. In yet other embodiments, the patching instructions are trap instructions, and the appropriate memory location for the referenced patch handler is determined from a return Program Counter map. Thus, in

these embodiments the patching of compiled executable file 132 is performed without recompiling, relinking or rewriting the file.

[0038] In addition to inserting patching instructions, Patching Subsystem 136 must also create appropriate patch handlers 184 so that the patch statements are exhibited when flow of execution is transferred to the patch handler. Each patch handler 184 will first save the current state of execution (*e.g.*, register and stack values) if necessary, and then include the code fragment executables and/or trace requests to be executed for the patch. If the patch is to be added to compiled executable code 132 rather than substituting for the compiled instruction replaced by the patching instruction, the replaced instruction must also be added to the patch handler so that it will be executed. For example, if the patch is to be added after the replaced instruction, then the replaced instruction would be added at the beginning of the patch handler before any of the patch statements. Alternately, if the patch is to be added before the replaced instruction, the replaced instruction would instead be added after the patch statements. Finally, the patch handler will, if necessary, restore the saved state of the computer and will then return the flow of execution to the instruction in the compiled executable code after the patching instruction.

[0039] Similarly to the created code fragment executables 186, each patch handler 184 will be uniquely identified in some way so that it can be referenced by its associated patching instruction. In one embodiment, a patch handler will always be loaded at a specific location in memory 180 on target computer system 150, and thus the patching instruction can explicitly indicate that memory location when the patching instruction is added to the patched compiled executable file 182 by the Patching Subsystem 136. This memory 180 location can be identified at the time the patch handler 184 is created if the Patching Subsystem 136 performs memory management for the target computer system 150 in which it can allocate and deallocate blocks of memory. Alternately, the Patching Subsystem 136 could request that the target computer system 150 allocate a block of memory 180 and provide the memory location information to the Patching Subsystem 136. In an alternate embodiment, patch handlers 184 are loaded into different locations of memory 180 with each execution of the patched executable code file 182. In that embodiment, Link Translation Table 188 can be used to map a unique identifier for each patch handler 184 to its current location in memory 180. In a similar manner, code fragment executables 186 can be stored with patch handlers 184 and loaded into the memory block for the patch handlers, or can instead be loaded into separate locations in memory 180. If loaded separately in memory 180, a patch handler can merely reference a unique identifier for a code fragment executable, and Link Translation Table 188 can be used to identify the current memory location at which to access the code fragment executable. In another embodiment, patch handlers 184 and code fragment executables 186 are loaded into different locations of memory 180 with each execution of the patched executable code file 182, but rather than using Link Translation Table 188 the patched compiled executable file 132 is modified before each execution so that identifiers are replaced with the current memory location for the associated item.

[0040] When the patched executable code file 182 is to be executed, the Patch Interpreter Subsystem 181 on the target computer system 150 controls the execution. In particular, the Patch Interpreter Subsystem 181 ensures that compiled instructions are executed as compiled native code on target computer system 150, while patch statements (if stored in machine-independent form) are interpreted. The Patch Interpreter Subsystem 181 also uses Link Translation Table 188 to resolve references to symbols, modules, code fragment executables, and patch handlers as necessary. Thus, if a patching instruction merely references the unique identifier for a patch handler 184, the Patch Interpreter Subsystem 181 will identify the memory location of the patch handler and transfer flow of execution there rather than trying to execute the patching instruction as native code. As trace requests are encountered during execution, the specified information for the trace request is stored in an appropriate manner, such as in Trace Log 189. The Patch Interpreter Subsystem 181 performs its interpreted evaluation in a manner so as to ensure that variables and expressions are evaluated in the context of the appropriate current variable scope. In one mode, the Patch Interpreter Subsystem 181 works independently without external influence. In an alternate mode, a user on the host computer system 100 can use the Command Subsystem 134 to interact with Patch Interpreter Subsystem 181 and to control execution of the patched executable file 182.

[0041] In one embodiment, the target computer system 150 is an embedded computer system, the Code Patcher system 133 is part of a debugger on the distinct host computer system 100, and the compiled executable file 132 is a terminal emulator to be executed on the embedded system. In this embodiment, the Code Patcher system 133 allows a user to perform normal debugging functions (*e.g.*, single stepping through a compiled program) but also allows the user to interactively modify an existing compiled executable code file in a permanent manner without requiring a compile/link/transfer/debug cycle. The user can also manipulate the patches in a variety of ways in this embodiment. For example, a patch can be added to the compiled executable code file but can be disabled. In this situation, the functionality added by the patch is not executed while the patch is disabled, either by replacing the patch with the original patched instruction or by indicating to the Patch Interpreter Subsystem 181 to not execute the patch statements. In addition, different patches can be grouped together and each group can be manipulated

in the same manner as a single patch. For example, all patches in a group can be enabled or disabled, or can be saved for later use. Patches can also be qualified with conditions such that the patch will be executed only when the conditions are true at the time of execution.

[0042] In addition to significantly reducing the time needed to create source code, the Code Patcher system 133 can be used for a variety of other purposes. For example, an executable code file may have been incomplete when compiled. In this situation, references to symbols (*e.g.*, variables or functions) may have been stubbed at the time of compilation. The Code Patcher system 133 can be used to replace the stubs with references to code fragment executables. In addition, additional functionality can be added even to compiled executable code files that lack errors. For example, a graphical user interface could be added to an executable code file with only a command-line interpreter, or new functionality can be added to handle inputs or situations not anticipated at the original time of compilation. Those skilled in the art will appreciate that any such type of functionality can be added using the Code Patcher system 133. Those skilled in the art will also appreciate that computer systems 100 and 150 are merely illustrative and are not intended to limit the scope of the present invention. The computer systems may contain additional components or may lack some illustrated components. Accordingly, the present invention may be practiced with other computer system configurations, including a single computer system.

[0043] As an illustrative example of patching compiled executable code, consider Figures 2A and 2B. Figure 2A illustrates sequences of compiled instructions from a compiled executable code file. For readability, a corresponding source code line is shown for each instruction. Those skilled in the art will appreciate that the particular source code lines are merely illustrative. Figure 2B illustrates the same compiled executable code file after various patches have been added. In particular, five patches, including a code fragment executable module, have been added to the original compiled executable code file.

[0044] Patch 1 demonstrates a patch that adds a trace request before instruction N. As is shown in Figure 2B, instruction N is replaced in the compiled executable code file with a patching instruction that indicates to transfer flow of execution to a patch handler for Patch 1. When execution is transferred to the patch handler for Patch 1, the patch handler first saves the current state of the target system on which execution is occurring. The patch handler then issues a trace request for a designated string of information. The original patched instruction, instruction N, is then executed in the patch handler. Thus, Patch 1 executes the trace request in addition to, rather than substituted for, the replaced instruction N. After executing instruction N, the original state of the computer system is then restored before the flow of execution is returned to instruction N+1. Those skilled in the art will appreciate that an execution such as the trace request can be performed in a variety of ways, such as by writing the specified information to a file or by sending the information to another computer, such as in a message object or as text in an email message. Alternately, trace information can be output using any other communication means (*e.g.*, pager, cellular phone, display device, etc.). Note that although the trace request in Patch 1 did not include any variables or expressions to be evaluated, instruction N will require the evaluation of the variables a and c. This evaluation will occur in the context of the appropriate current variable scope.

[0045] Patch 2 reflects a patch which replaces instruction N+3 with multiple patch statements, including both a trace request and a code fragment executable. Patch 2 also demonstrates a patch that substitutes for an existing instruction rather than adding patch statements, and thus existing instruction N+3 will be not executed in the patched executable code file. When flow of execution is transferred to the patch handler for Patch 2, the patch handler first saves the current target computer system state. The-patch handler then executes the trace request and the code fragment executable as shown. Note that this trace request will require the evaluation of variable a, which will be performed in the context of the appropriate current variable scope. The code fragment executable will be executed in an interpreted manner. After executing the various instructions in the patch handler, the original computer system state is restored before returning flow of execution to instruction N+4.

[0046] As is shown in Figure 2B, the code fragment executable module labeled FN2 has also been added as part of the patch process. Patch 3 adds patch statements before instruction N+6, including a call to the newly created module. Patch 3 also illustrates that multiple code statements can be executed within a patch handler (limited only by available memory on the target computer system). After flow of execution transfers to the patch handler for Patch 3, the appropriate computer system state is first saved. Next, the code fragment executable for the code statement 'b=b-1' is executed in an interpreted manner. The next patch statement calls module FN2 with the variable b as a parameter. Flow of execution will then transfer to the code fragment executable for module FN2, at which time instructions M, M+1, and M+2 will be executed in sequence in an interpreted manner. Upon execution of the return statement in instruction M+2, the flow of execution will return to instruction N+6 in the patch handler for Patch 3. After restoring state for Patch 3, the flow of execution will return to instruction N+7 in the patched compiled executable code file. Those skilled in the art will appreciate that each of the patch handlers can return flow of execution to the compiled instructions in a variety of ways, such as using a return statement or explicitly using the memory location of the next compiled instruction. In addition,

in one embodiment each code statement has a separate code fragment executable, while in another embodiment the various code statements in a patch handler are combined into a single code fragment executable.

[0047] Finally, Patch 4 illustrates that compiled functions can be patched and traced in a manner similar to that of any other compiled instruction. For example, if a user desires to trace all invocations of the function FN1, Patch 4 can be added before the first instruction of the function, instruction P. The trace requests in Patch 4 illustrate that when the variable X is evaluated, it will be evaluated using the current variable scope of function FN1, but that the global variable Y will be evaluated using the appropriate variable scope for Y, which may be the entire program rather than the local scope of function FN1. Those skilled in the art will appreciate that a variety of other patches are possible

[0048] Referring now to Figures 3A and 3B, illustrative examples of code fragment executables stored in an executable machine-independent non-compiled format are shown. In particular, the exemplary machine-independent format shown are parse trees. In Figure 3A, the source code statement 'A = A + (B*3)' is shown, along with a corresponding parse tree that encodes this source code statement. The original source code line can be reconstructed by using depth-first search along the parse tree, using non-leaf nodes only after the first child branch has been used. Similarly, Figure 3B illustrates a possible parse tree for the source code function Swap as shown. As the parse tree is read from left to right and upwards from the leaf nodes, the appropriate order of statements for the Swap function is retrieved. Those skilled in the art will appreciate that a parse tree can be represented using different formats, and that parse trees are only one possible format for encoding executable machine-independent code fragment executables.

[0049] Figure 4 is an exemplary flow diagram of an embodiment of the Command Subsystem routine 400. The Command Subsystem routine will receive a command from a user to either patch a compiled executable code file such as code file 132 or to execute an already patched executable code file such as patched code file 182. In an alternate embodiment, patches can be added to a compiled executable code file that already contains patches (and can even patch already patched instructions), as well as to a compiled executable code file without patches. If the user is currently patching a compiled executable code file, the routine will accept a variety of trace requests and code fragments to be added to the code file, will combine multiple patches if they are to be added at the same compiled instruction in the compiled executable code file, and will notify the Patching Subsystem 136 (Figure 1) and the Code Fragment Executables Subsystem 138 (Figure 1) of the patches. If the user is executing an already patched executable code file, the routine can receive commands, relay them to the Patch Interpreter Subsystem 181 (Figure 1), and receive and display to the user the results of the execution of the command. In one embodiment in which the Command Subsystem 134 (Figure 1) provides interactive control over a patched executable code file, a user can determine whether or not to view the source code for patches (*e.g.*, not viewing source code for disabled patches).

[0050] The Command Subsystem routine begins at step 405 where a command is received from a user. The routine continues in step 410 to determine if the command is to patch a compiled executable code file. If so, the routine continues to step 415 where it receives an indication of the executable code file to be patched. The routine then continues in step 420 to load information from the symbol table of the indicated executable code file, thus allowing the source code lines corresponding to the compiled executable code instructions to be displayed to the user. The routine then continues to step 425 where a request is received from the user that is related to patching the compiled executable code file. The routine continues in step 430 to execute the Process Patch Request subroutine to process the request. As is explained in greater detail with relation to Figure 5, possible patch requests include adding various trace requests or code fragments, grouping and degrouping patches, and manipulating previously created patches. After the patch request has been processed in step 430, the routine continues in step 435 to determine if there are more patch requests from the user. If so, the routine returns to step 425 to receive the next patch request.

[0051] If there are not more patch requests from the user, the routine continues to step 440 to sequence patches together when they will replace the same compiled instruction. Since any given compiled instruction in the compiled executable code file can only be replaced with a single patching instruction to a patch handler, multiple patches that have been associated with a single compiled instruction must be sequenced together. Additional steps may also need to be taken to ensure that all patches are executed and that the correct order of patch execution is followed. In the illustrated embodiment, multiple patches associated with a single compiled instruction are placed in a single patch handler for the patching instruction that will replace the compiled instruction. Alternately, multiple patch handlers could be used, with each patch handler transferring flow of execution to the next patch handler in sequence and with the last patch handler returning the flow of execution to the compiled instruction following the patching instruction. In either case, a sequence in which to execute the multiple patches must be selected. For example, all trace requests could be executed either before or after all code fragments. In the illustrated embodiment, the user indicates the sequence of patches when the patches are created.

[0052] After step 440, the routine continues to step 445 to notify the Patching Subsystem 136 of the patch sequences to be added to the compiled executable code

file, including sequences of a single patch. The routine then continues to step 450 to notify the Code Fragment Executables Subsystem 138 of the code fragments, including created modules. that have been added as patches to the compiled executable code file. The routine continues in step 455 to determine if there are more commands from the user. If so, the routine returns to step 405 to receive a user command, and if not, the routine ends at step 499.

[0053] If it is instead determined in step 410 that the received user command is not to patch a compiled executable code file, the routine continues to step 460 where an indication of a patched compiled executable code file is received. This patched code file will be executed on an indicated target computer system. In the illustrated embodiment, the patched code and any associated patch handlers and code fragment executables will have been transferred to the indicated target system when the patching occurred. Alternately, the transferring could occur after step 460. The routine next continues to step 465 to determine if the patched code file is to be executed with interactive control by the user. If the patched executable code file is not to be executed interactively, the routine continues to step 470 where the Patch Interpreter Subsystem 181 is notified to execute the patched executable code file without waiting for interactive commands. If instead it is determined in step 465 that the patched executable code file is to be executed interactively, the routine continues instead to step 475 where the Patch Interpreter Subsystem 181 is notified to execute the patched executable code file in an interactive execution mode.

[0054] The routine next continues to step 480 to receive an execution-related command from the user. Such commands could include setting or removing break points, checking the current values of variables and expressions, or stepping through the execution a single line at a time. In the illustrated embodiments, the manipulation of patches does not occur during interactive execution, but such functionality could be provided in an alternate embodiment. In step 485, the routine notifies the Patch Interpreter Subsystem 181 of the command, and in step 490 the results of the command execution are received from the Patch Interpreter Subsystem 181 and are displayed to the user. The routine then continues to step 495 to determine if there are more interactive execution commands from the user. If so, the routine returns to step 480 to receive the next user command. After step 470, or if it is determined in step 495 that there are not more interactive execution commands, the routine continues to step 455 to determine if there are more user commands. Those skilled in the art will appreciate that the Command Subsystem routine can be implemented in a variety of ways. For example, other top-level user commands than patching compiled executable code files and executing patched compiled executable code files could be processed by the routine. Alternately, separate routines could be used to patch a compiled executable code file and to execute a patched compiled executable code file.

[0055] Figure 5 is an exemplary flow diagram of an embodiment of the Process Patch Request subroutine 430. The Process Patch Request subroutine receives a user request related to patching the compiled executable code file, determines the type of patch request, and satisfies the patch request. In the illustrated embodiment, the user indicates source code lines corresponding to compiled instructions in the compiled executable code file for patch requests, and the Code Patcher system 133 selects the appropriate compiled instruction for the indicated source code line when performing the request.

[0056] The subroutine begins at step 505 where it is determined if the patch request is to add a trace request to indicate when a source code line is executed. If so, the subroutine continues to step 510 to add the trace request to an appropriate patch handler, which will be referenced by a patching instruction that replaces a compiled instruction corresponding to the indicated source code line. If it is instead determined that the trace request is not to trace a source code line, the subroutine continues to step 515 to determine if the patch request is to add a trace request to indicate when a function in the compiled executable code file is executed. If so, the subroutine continues to step 520 to add a trace request to a patch handler for the first instruction of the indicated function. if the patch request is not to trace a function, the subroutine continues to step 525 to determine if the patch request is to add a trace request for a specified expression at a particular source line. If so, the subroutine continues to step 530 to add a trace request for the expression to a patch handler at the indicated source line. If the patch request is not to trace a specified expression, the subroutine instead continues to step 535 to determine if the patch request is to add a specified code fragment to the compiled executable code file. If so, the subroutine continues to step 540 to add the specified code fragment executable to a patch handler at the compiled instruction corresponding to an indicated source line. The user can also indicate whether the patch is to substitute for the instruction or to be added in addition to the instruction, and when the patch is added in addition to the instruction whether the patch is to be executed before or after the source code line.

[0057] If it has been determined in steps 505, 515, 525, and 535 that the patch request is not to add to the compiled executable code file patch statements that require patch handlers, the subroutine continues to step 545 to determine if the patch request is to create a code fragment module. If so, the subroutine continues to step 550 where an invokable code module with multiple source code statements is specified by the user, and an associated code fragment executable module is created for the patched executable code file. If it is determined in step 545 that the patch request is not to create a code module, then the subroutine continues to step 555 to

determine if the patch request is to group patches together or to degroup previously grouped patches into individual patches. If so, the subroutine continues to step 560 to perform the indicated grouping or degrouping. If the patch request is not to group or degroup patches, the subroutine continues to step 565 to determine if the patch request is to manipulate a patch or patch group. If so, the subroutine continues in step 570 to manipulate the patch or the patches in a patch group as indicated. After step 510, 520, 530, 540, 550, 560, or 570, or if it is determined in step 565 that the patch request does not manipulate a patch or a patch group, the subroutine continues to step 590 where it returns.

[0058] Figure 6 is an exemplary flow diagram of an embodiment of the Patching Subsystem routine 600. The Patching Subsystem routine is notified by the Command Subsystem routine when patch sequences are to be added or manipulated for a compiled executable code file. The Patching Subsystem routine then creates the necessary patch handlers, patches the compiled executable code file in the appropriate manner, and loads the patch handlers and the patched compiled executable code file onto a target system. The routine begins at step 605 where one or more patch sequences are received for a compiled executable code file. This notification from the Command Subsystem 134 can also indicate that previously added patches have been manipulated (*e.g.*, removed or disabled). The routine continues in step 610 to select the next received patch sequence to be added or manipulated, beginning with the first received patch sequence. The routine then continues to step 615, where an appropriate patch handler is created if the selected patch sequence indicates that a patching instruction is to be added to the compiled executable code file. Alternately, if an existing patch is being changed, an existing patch handler can be accordingly modified or a new patch handler can be created to replace an existing patch handler. For example, if a previously added patch is being removed, the original patched instruction can be reinserted in the compiled executable code file in place of the patching instruction. Alternately, if a patch handler is being disabled, the first instruction of the patch handler could be modified to cause a jump to the end of the patch handler or to the instruction following the patching instruction. Similarly, if a patch is being qualified, the patch handler for that patch can be modified to include evaluation statements to determine if the specified conditions are true.

[0059] After step 615, the routine continues to 620 to, if a patching instruction needs to be added, replace the appropriate compiled executable code instruction with a patching instruction (*e.g.*, a jump statement) to the created patched handler. If an existing patch handler is merely being modified, it will not be necessary to replace a compiled instruction with a patching instruction since the existing patching instruction will still be effective. While the illustrated embodiment uses a jump instruction for patching instructions, those skilled in the art will appreciate that a patch handler can be invoked in a variety of ways, such as by using an interrupt mechanism or by including an identifier that will be resolved at execution time to indicate an appropriate location in memory on the target system. In addition, those skilled in the art will appreciate that a compiled transfer statement instruction could be used as a patching instruction. After step 620, the routine continues in step 625 to determine if there are more patch sequences to be selected. If so, the routine returns to step 610 to select the next patch sequence, and if not, the routine continues to step 630.

[0060] After patch handlers have been created or modified for each patch sequence and patching instructions have been added to the compiled executable code file, the routine in step 630 loads the patched executable code file onto the target system. The routine then continues in step 635 to load any created and/or modified patch handlers onto the target system. In the illustrated embodiment, the Patching Subsystem 136 does not perform memory management for target computer system 150, so the memory locations on the target system in which the created patch handlers will be loaded will not be known until execution time. Thus, the patching instructions inserted into the compiled executable code file merely reference the appropriate patch handler, with the reference to the appropriate memory location to be resolved at execution time. This type of reference resolution allows linking to be delayed until execution time. Thus, after step 635, the routine continues in step 640 to update a Link Translation Table on the target system to reflect the unique identifiers for the created patch handlers. After step 640, the routine continues to step 645 to determine if there are more patch sequences to receive. If so, the routine returns to step 605 to receive patch sequences, and if not the routine continues to step 690 and ends.

[0061] Figure 7 is an exemplary flow diagram of an embodiment of the Code Fragment Executables Subsystem routine 700. The Code Fragment Executables Subsystem routine receives notifications from the Command Subsystem 134 when code fragments are to be added to the compiled executable code file, creates machine-independent non-compiled code fragment executables that can be interpreted by the Patch Interpreter Subsystem 181, and loads the created code fragment executables onto the target system. The routine begins in step 705 where code fragments to be added to the compiled executable code file are received from the Command Subsystem. The routine continues to step 710 to select the next code fragment, beginning with the first code fragment. In step 715, the routine then creates a code fragment executable for the selected code fragment in a machine-independent format not specific to the target computer. In the illustrated embodiment, the code fragment executables are stored in parse tree format. The routine then continues in step 720 to determine if there are more code fragments to process. If so, the routine returns to step 710 to select the next code frag-

ment, and if not the routine continues to step 725.

[0062] After the code fragment executables have been created, the routine in step 725 loads the created code fragment executables onto the target system. The routine then continues to step 730 to update the Link Translation Table on the target system to reflect the references for the created code fragment executables. For example, if invokable code modules have been created for the patched compiled executable code file, an entry in the Link Translation Table can be used to resolve at execution time a reference to a code module, allowing flow of execution to be transferred to the appropriate memory location where the code fragment executable for the module is loaded in memory. Similarly, if variables have been defined and made available to other portions of the code outside a self-contained variable scope (*e.g.*, global variables), the Link Translation Table can be used to resolve references to those variables by other portions of the code. Finally, if the code fragment executables are loaded into memory separate from the patch handlers that will reference those code fragment executables, then the Link Translation Table can be used to resolve the references for each such code fragment executable.

[0063] Those skilled in the art will appreciate that if the memory locations for code fragment executables are known at the time that the patches and patch handlers are created, it may not be necessary to use a Link Translation Table to resolve those references. For example, if code fragment executables are included with the patch handlers which reference them, then each such code fragment executable would not need a unique identifier that could be resolved through the use of the Link Translation Table. After step 730, the routine continues in step 735 to determine if there are more code fragments to receive. If so, the routine returns to step 705, and if not the routine ends at step 790.

[0064] Figure 8 is an exemplary flow diagram of an embodiment of the Patch Interpreter Subsystem routine 800. The Patch Interpreter Subsystem routine receives an indication of a patched executable code file to be executed, loads the patched compiled executable code file as well as associated patch handlers and code fragment executables into memory, executes the non-patched compiled instructions as native code for the target system, and executes the patch statements in an interpreted manner. Those skilled in the art will appreciate that in an alternate embodiment, before execution time the Patch Interpreter Subsystem 181 or another subsystem could compile the target-independent code fragment executables into compiled instructions that are native to the particular target system. In this situation, the Patch Interpreter Subsystem 181 would only be needed to resolve references, such as for patch, handlers and code fragment executable modules, when such references are included in the patched compiled executable code file.

[0065] The routine begins in step 805 where an indication is received of the patched executable code file to be executed. In the illustrated embodiment, this indication is received from the Command Subsystem 134 on the host computer system. In an alternate embodiment, the Patch Interpreter Subsystem 181 could receive instructions directly from a user on the target computer system if the target system included the necessary input/output devices. The routine then continues to step 810 to load the patch handlers and code fragment executables associated with the patched compiled executable code file into memory if they are not already stored in memory. The routine then continues to step 815 to ensure that the Link Translation Table reflects the current memory addresses for the code fragment executables and the patch handlers. After step 815, the routine continues to step 820 where the patched compiled executable code file is loaded into memory if it is not already stored in memory. The routine then continues to step 825 to execute the Execute Patched Executable Code subroutine. The routine then continues to step 830 to determine if there are more patched compiled executable code files to execute. If so, the routine returns to step 805, and if not, the routine ends at step 890. Those skilled in the art will appreciate that in an alternate embodiment, the Patch Interpreter Subsystem routine could execute multiple different patched executable code files at the same time.

[0066] Figure 9 is an exemplary flow diagram of an embodiment of the Execute Patched Executable Code subroutine 825. The Execute Patched Executable Code subroutine receives and executes user commands if an interactive execution of the patched compiled executable code file is in process. When a non-patched compiled instruction is to be executed, it is executed as native code. For each patch statement, the subroutine first determines if the statement is to be executed and then resolves link identifiers if necessary. The subroutine then satisfies trace requests by writing the specified trace information to a trace log on the target system, executes code fragment executables by interpreting them, and notifies the Code Patcher system 133 of the execution status after the instruction has been processed.

[0067] The subroutine begins at step 905 where it is determined if the current execution of the patched compiled executable code file is to be done in an interactive mode. If so, the routine continues in step 910 to receive a user command. The subroutine then continues to step 915 to determine if the received command is to execute the current instruction in the patched executable code file. If not, the subroutine continues to step 920 to execute the user command, and then continues to step 925 to notify the Code Patcher system 133 of the results of executing the command. For example, a user may wish to see a current value for a variable or to specify a break point at a spot in the patched compiled executable code file. After such a break point has been set, the user may indicate to continue execution in a non-interactive mode

until a break point is reached, thus causing execution to stop and returning execution of the patched executable code file to an interactive mode. After step 925, the subroutine returns to step 905.

[0068] If it was instead determined in step 905 that the execution of the patched executable code file is not currently in interactive mode, or if it was determined in step 915 that the user command was to execute the current instruction, the subroutine continues in step 930 to select the next instruction in the patched executable code file, beginning with the first instruction. The subroutine then continues to step 931 to determine whether the current instruction is to be currently executed. For example, the current instruction could be a patching instruction for a patch that is disabled or that is qualified with a condition that is not currently met in the current environment. If the instruction is not to be currently executed, the subroutine continues to step 970 where the Code Patcher system 133 will be notified of the status of execution of the current instruction if execution is occurring in an interactive mode. Those skilled in the art will appreciate that when an entire patch handler consisting of multiple patch statements has been disabled or is not qualified, the status of the patch handler can be indicated in a variety of ways. For example, a flag could be set when the patch handler was entered indicating that patch statements are not to be currently executed while the flag is set, with the flag being reset when the end of the patch handler is reached. Alternately, the patch handler could cause the flow of execution to skip any disabled patch statements.

[0069] If it is determined in step 931 that the current instruction is to be executed, the subroutine continues to step 932 to determine if the instruction is a compiled instruction. If so, the subroutine continues to step 933 to execute the current instruction in native mode on the target system, and then continues to step 970. If it is instead determined in step 932 that the current instruction is a non-compiled patch statement or patching instruction, the subroutine continues in step 935 to determine if the current instruction includes a link identifier reference that needs to be resolved. If so, the subroutine continues in step 940 to use the Link Translation Table to resolve the link identifier so that the current position in memory for the associated code can be accessed. After step 940, or if it is determined in step 935 that a link identifier is not present, the subroutine continues in step 945 to determine if the current instruction is a trace request. If so, the subroutine continues to step 950 to write the specified trace information to a trace log on the target system that can be retrieved by a user on the host computer system 100. If the trace request includes variable or expressions to be evaluated, they will be evaluated in the context of the current variable scope.

[0070] If it is instead determined in step 945 that the current instruction is not a trace request, the subroutine continues to step 965 to execute the code fragment executable by interpreting it within the context of the current variable scope. For example, a global variable X may be defined, with the current flow of execution first entering Function 1 with a variable X that is local to that function and then entering a Function 2 with a distinct local variable X for that function. If patch code within Function 2 requires the evaluation of variable X, its current variable binding within the scope of Function 2 will be used for the evaluation. Those skilled in the art will appreciate that the current variable scope can be determined in a variety of ways, such as by using symbol table information from the patched executable code file or the current variable block on the stack. After step 950 or 965, the subroutine continues at step 970 to notify the Code Patcher system 133 of the status of executing the current instruction if execution is occurring in interactive mode. The subroutine then continues to step 975 to determine if there are more instructions to be executed in the patched compiled executable code file. If so, the subroutine returns to step 905, and if not the subroutine returns in step 990.

[0071] From the foregoing it will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. Accordingly, the invention is not limited except as by the appended claims.

Claims

1. A method for modifying an executable file including a plurality of compiled instructions, the compiled instructions natively executable on a target computer but not on a source computer, the modifying performed under control of the source computer without recompiling or relinking or rewriting the executable file, the method comprising:

   under control of the source computer,

   (a) loading the executable file onto the target computer;
   (b) receiving a plurality of indications to modify the loaded executable file, each indication specifying one of the compiled instructions in the loaded executable file and at least one non-compiled executable instruction to be added to the loaded executable file;
   (c) modifying the loaded executable file by, for each indication,

   creating an instruction group including the specified at least one non-compiled executable instruction;
   converting the instruction group into an executable non-compiled format

not specific to the target computer; and
replacing the specified compiled instruction in the loaded executable file with an executable patch instruction associated with the converted instruction group, the patch instruction when executed to transfer flow of execution to the converted instruction group with which the patch instruction is associated; and

(d) loading the converted instruction groups into at least one memory area on the target computer, the at least one memory area distinct from a memory area on the target computer containing the loaded executable file; and

under control of the target computer, executing the modified loaded executable file by, as the flow of execution sequentially reaches each executable instruction,

(e) when the executable instruction is a non-compiled executable instruction from an instruction group, interpretively evaluating the executable instruction;
(f) when the executable instruction is a patch instruction, transferring flow of execution to the loaded associated converted instruction group; and
(g) when the executable instruction is a compiled instruction, natively executing the compiled executable instruction,

so that additional functionality is exhibited when the modified loaded executable file is executed on the target computer.

2. The method of claim 1 wherein after performing steps (a)-(g) a first time, steps (b)-(g) are performed a second time without performing step (a) a second time such that the modified loaded executable file is further modified and then executed without reloading the executable file.

3. A computer-implemented method for adding instructions to be executed to an executable compiled file without recompiling or rewriting the file, the method comprising:

receiving an indication to modify the compiled file, the indication specifying a compiled instruction in the compiled file and at least one executable instruction to be added;
modifying the compiled file by,

creating a patch group including the specified at least one executable instruction; and
replacing the specified compiled instruction in the compiled file with a patch instruction such that flow of execution will transfer to the created patch group upon execution of the patch instruction; and

storing separate from the patch group the modified compiled file including the patch instruction,

so that instructions in the patch group will be executed when the modified compiled file is later executed.

4. The method of claim 3 wherein the modified compiled file is executed on a separate computer, and including:

supplying to the separate computer at least one direction related to the execution of the modified compiled file; and
displaying received results of performance of the supplied direction.

5. The method of claim 3 wherein without removing the patch instruction from the modified compiled file, the patch group is disabled such that the instructions in the patch group are not executed when the modified compiled file is executed.

6. A computer-implemented method for executing a modified compiled file including a plurality of original compiled instructions and at least one patch instruction in place of an original compiled instruction, each patch instruction when executed to transfer flow of execution to an associated patch group including a plurality of instructions, the method comprising:

loading the modified compiled file and the associated patch groups into memory; and
executing the modified compiled file by, as the flow of execution reaches each executable instruction,

when the executable instruction is an original compiled instruction, natively executing the executable instruction;
when the executable instruction is one of the patch instructions transferring flow of execution to the associated patch group for the patch instruction; and
when the executable instruction is one of the plurality of instructions from a patch group, interpretively evaluating the executable instruction.

7. A computer system for adding instructions to be executed to an executable compiled file without recompiling or rewriting the file, comprising:

a command subsystem for receiving indications to modify the compiled file, each indication specifying a compiled instruction in the compiled file and at least one executable instruction to be added; and
a patching subsystem for modifying the compiled file by creating for each received indication an associated patch group including the specified at least one executable instruction, and by replacing the specified compiled instruction in the compiled file with a patch instruction such that flow of execution will transfer to the associated patch group upon execution of the patch instruction.

8. A computer-readable medium containing instructions for controlling a computer system to add instructions to be executed to an executable compiled file without recompiling the file, by:

receiving an indication to modify the compiled file, the indication specifying a compiled instruction in the compiled file and at least one executable instruction to be added; and
modifying the compiled file by,

creating a patch group including the specified at least one executable instruction; and
replacing the specified compiled instruction in the compiled file with a patch instruction such that flow of execution will transfer to the created patch group upon execution of the patch instruction.

9. The computer-readable medium of claim 8 wherein the computer system is further controlled by, after the modifying of the compiled file executing on a target computer the modified compiled file by, when an instruction to be executed is the patch instruction, indicating an instruction in the created patch group as a next instruction to be executed.

10. The computer-readable medium of claim 9 wherein the computer system is further controlled by, after the executing of the modified compiled file, performing the modifying and the executing again without reloading the compiled file into memory.

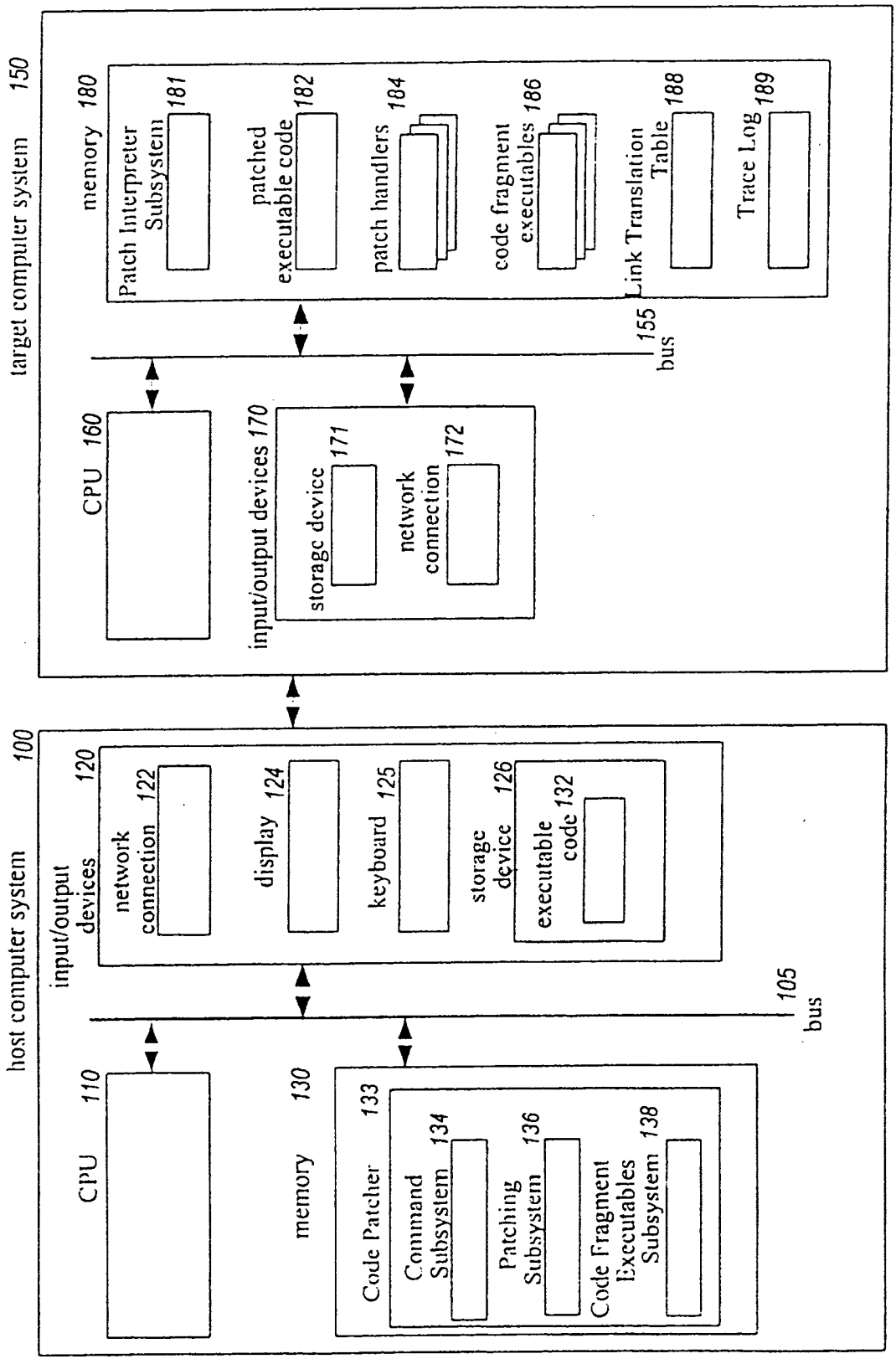

target computer system 150
memory 180
Patch Interpreter Subsystem 181
patched executable code 182
patch handlers 184
code fragment executables 186
Link Translation Table 188
Trace Log 189
bus 155
CPU 160
input/output devices 170
storage device 171
network connection 172
host computer system 100
input/output devices 120
network connection 122
display 124
keyboard 125
storage device 126
executable code 132
bus 105
CPU 110
memory 130
Code Patcher 133
Command Subsystem 134
Patching Subsystem 136
Code Fragment Executables Subsystem 138

Fig. 1

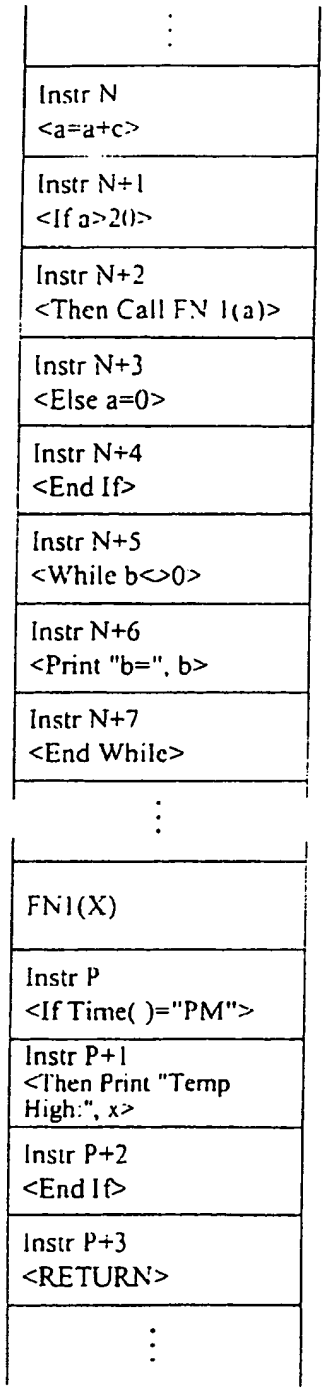

Instr N
<a=a+c>
Instr N+1
<If a>20>
Instr N+2
<Then Call FN 1(a)>
Instr N+3
<Else a=0>
Instr N+4
<End If>
Instr N+5
<While b<>0>
Instr N+6
<Print "b=", b>
Instr N+7
<End While>
FN1(X)
Instr P
<If Time( )="PM">
Instr P+1
<Then Print "Temp High:", x>
Instr P+2
<End If>
Instr P+3
<RETURN>

Fig. 2A

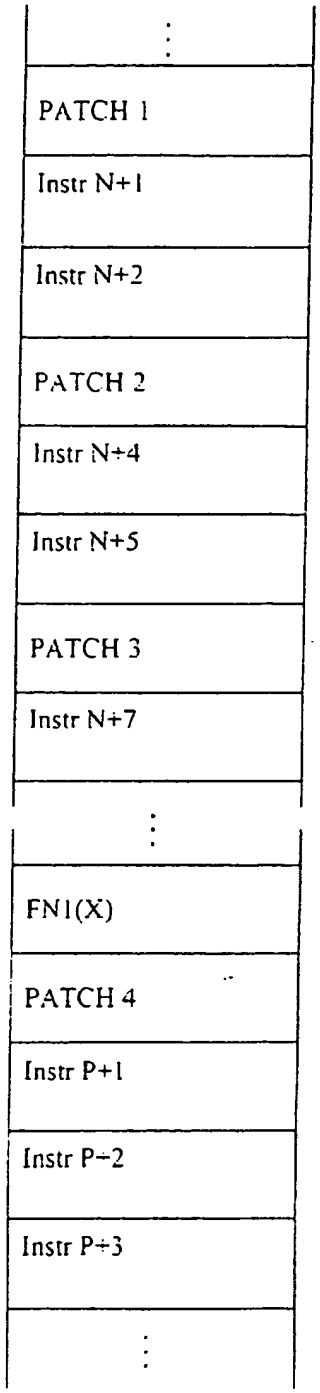
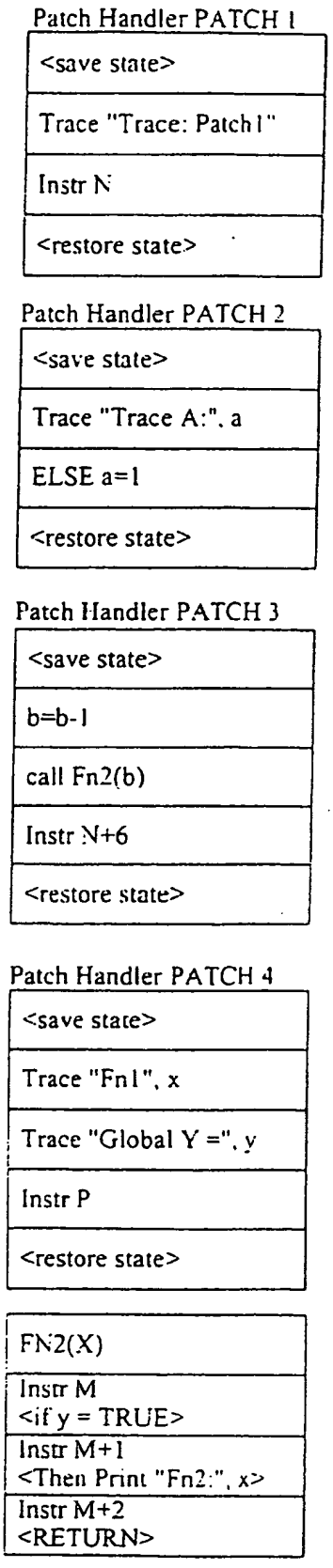

PATCH 1
Instr N+1
Instr N+2
PATCH 2
Instr N+4
Instr N+5
PATCH 3
Instr N+7
FN1(X)
PATCH 4
Instr P+1
Instr P+2
Instr P+3

Fig. 2B


Patch Handler PATCH 1
<save state>
Trace "Trace: Patch1"
Instr N
<restore state>
Patch Handler PATCH 2
<save state>
Trace "Trace A:", a
ELSE a=1
<restore state>
Patch Handler PATCH 3
<save state>
b=b-1
call Fn2(b)
Instr N+6
<restore state>
Patch Handler PATCH 4
<save state>
Trace "Fn1", x
Trace "Global Y =", y
Instr P
<restore state>
FN2(X)
Instr M
<if y = TRUE>
Instr M+1
<Then Print "Fn2:", x>
Instr M+2
<RETURN>

a = a + (b * 3)

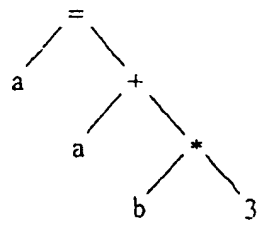


Fig. 3A

```
Function Swap (Ptr C, Ptr D)
  Global X = Value at Ptr C
  Value at Ptr C = Value at Ptr D
  Value at Ptr D = Global X
RETURN
```

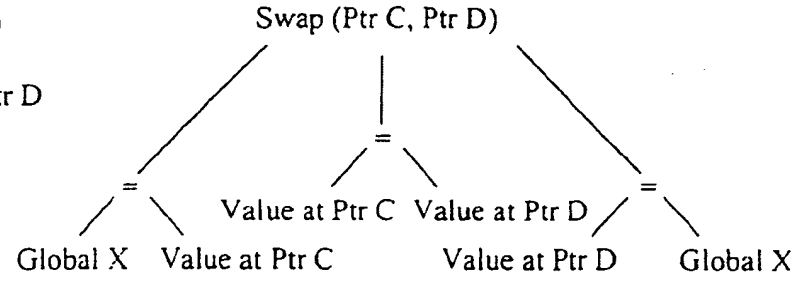


Fig. 3B

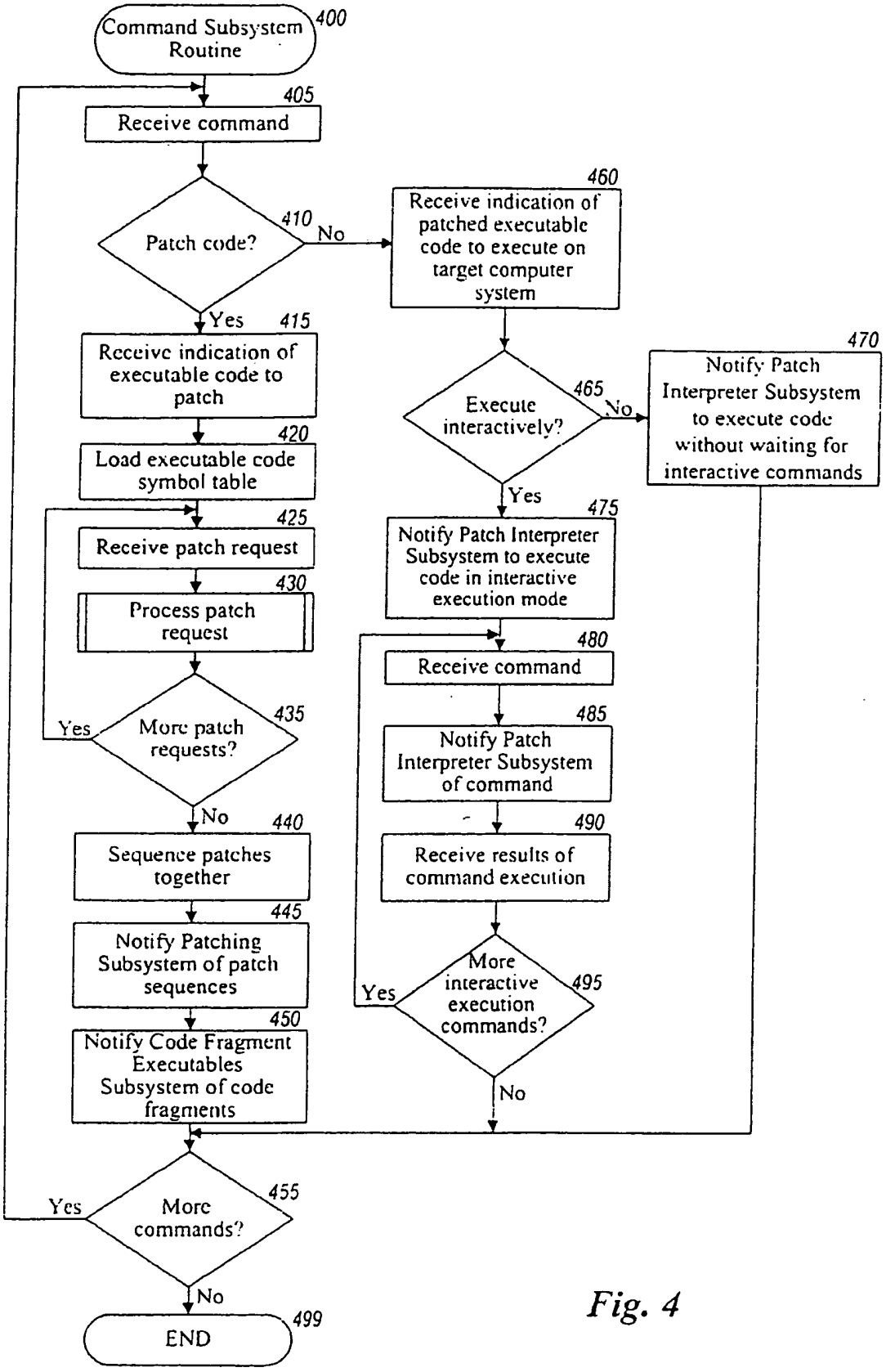

Command Subsystem Routine
400
Receive command
405
Patch code?
410
No
Yes
Receive indication of executable code to patch
415
Load executable code symbol table
420
Receive patch request
425
Process patch request
430
More patch requests?
435
Yes
No
Sequence patches together
440
Notify Patching Subsystem of patch sequences
445
Notify Code Fragment Executables Subsystem of code fragments
450
More commands?
455
Yes
No
END
499
Receive indication of patched executable code to execute on target computer system
460
Execute interactively?
465
No
Yes
Notify Patch Interpreter Subsystem to execute code without waiting for interactive commands
470
Notify Patch Interpreter Subsystem to execute code in interactive execution mode
475
Receive command
480
Notify Patch Interpreter Subsystem of command
485
Receive results of command execution
490
More interactive execution commands?
495
Yes
No

Fig. 4

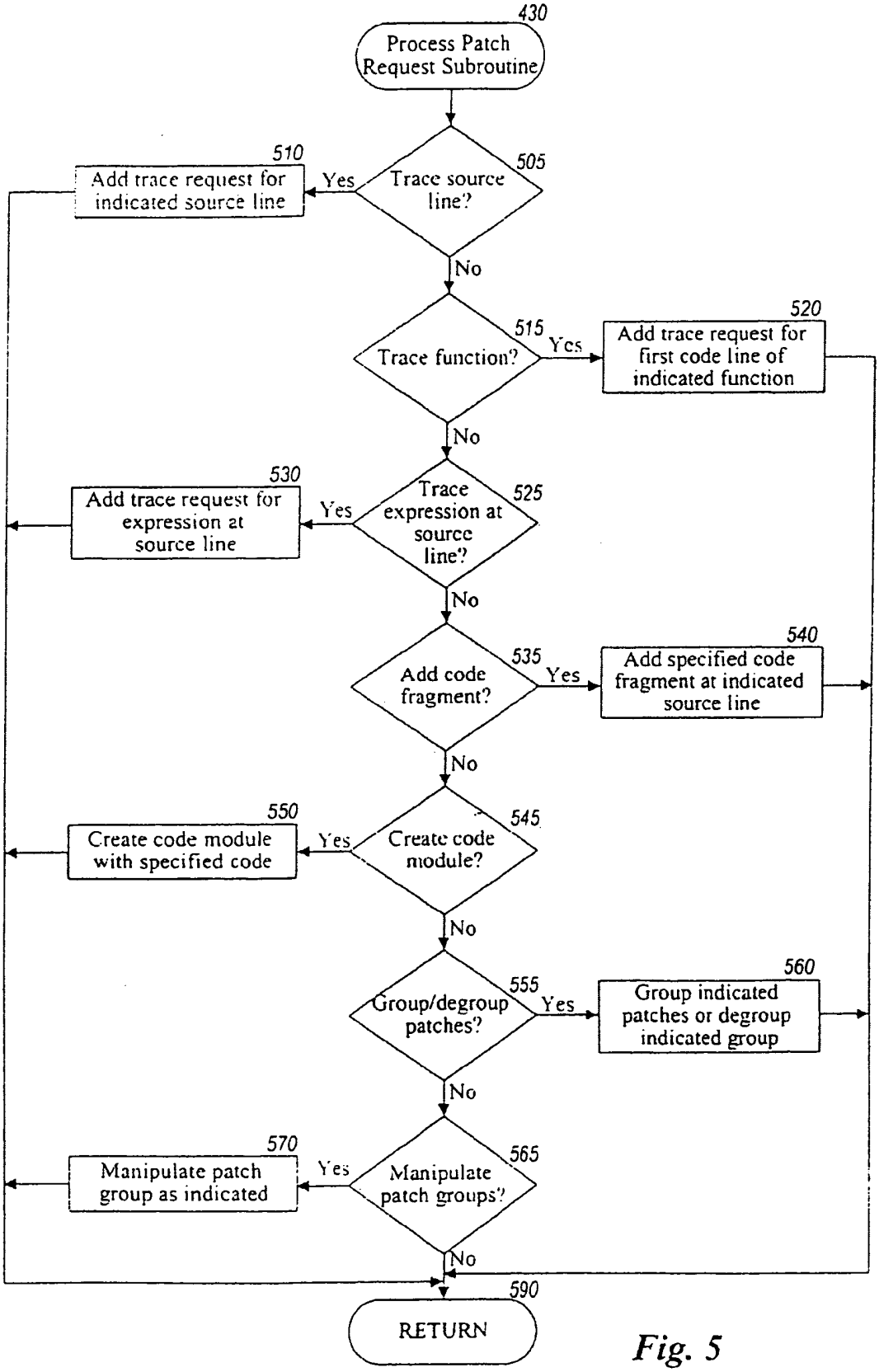

430
Process Patch Request Subroutine
505
Trace source line?
Yes
510
Add trace request for indicated source line
No
515
Trace function?
Yes
520
Add trace request for first code line of indicated function
No
525
Trace expression at source line?
Yes
530
Add trace request for expression at source line
No
535
Add code fragment?
Yes
540
Add specified code fragment at indicated source line
No
545
Create code module?
Yes
550
Create code module with specified code
No
555
Group/degroup patches?
Yes
560
Group indicated patches or degroup indicated group
No
565
Manipulate patch groups?
Yes
570
Manipulate patch group as indicated
No
590
RETURN

Fig. 5

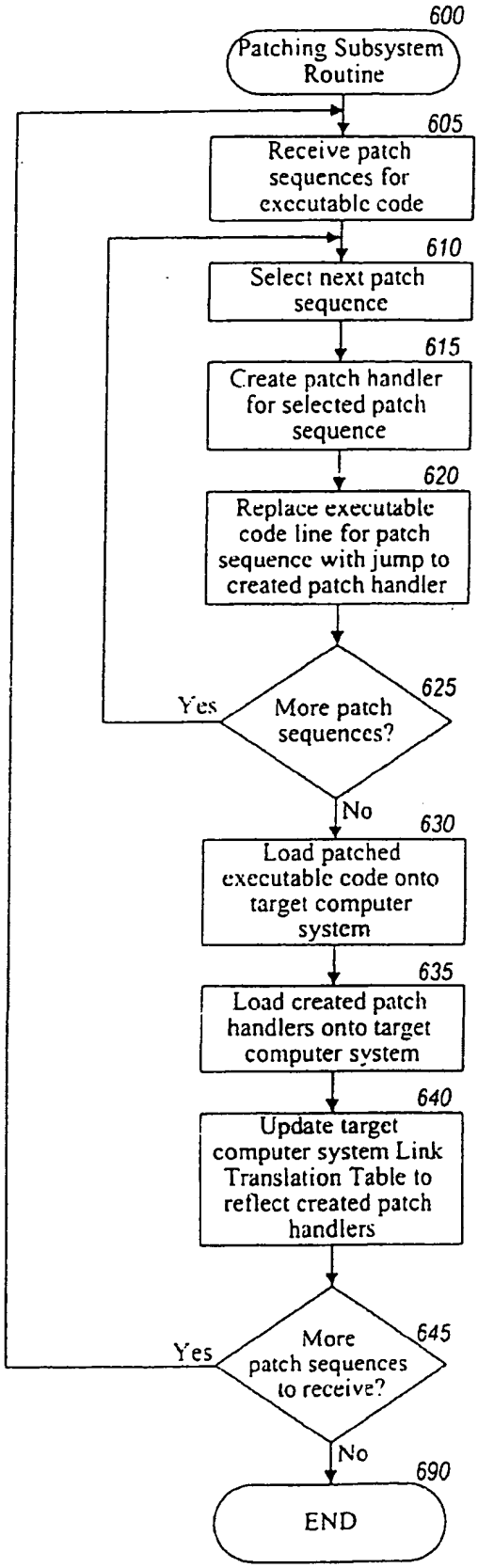

600
Patching Subsystem Routine
605
Receive patch sequences for executable code
610
Select next patch sequence
615
Create patch handler for selected patch sequence
620
Replace executable code line for patch sequence with jump to created patch handler
625
More patch sequences?
Yes
No
630
Load patched executable code onto target computer system
635
Load created patch handlers onto target computer system
640
Update target computer system Link Translation Table to reflect created patch handlers
645
More patch sequences to receive?
Yes
No
690
END

Fig. 6

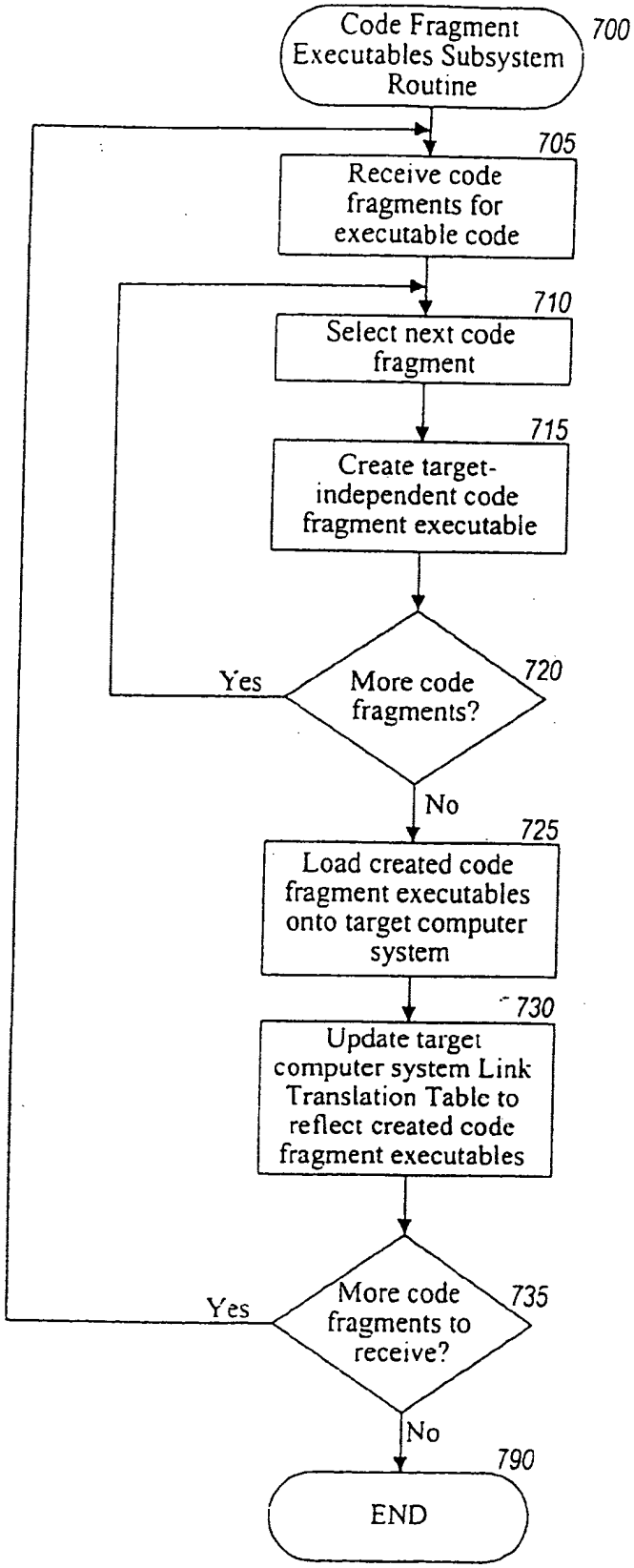

Code Fragment Executables Subsystem Routine
700
705
Receive code fragments for executable code
710
Select next code fragment
715
Create target-independent code fragment executable
720
More code fragments?
Yes
No
725
Load created code fragment executables onto target computer system
730
Update target computer system Link Translation Table to reflect created code fragment executables
735
More code fragments to receive?
Yes
No
790
END

Fig. 7

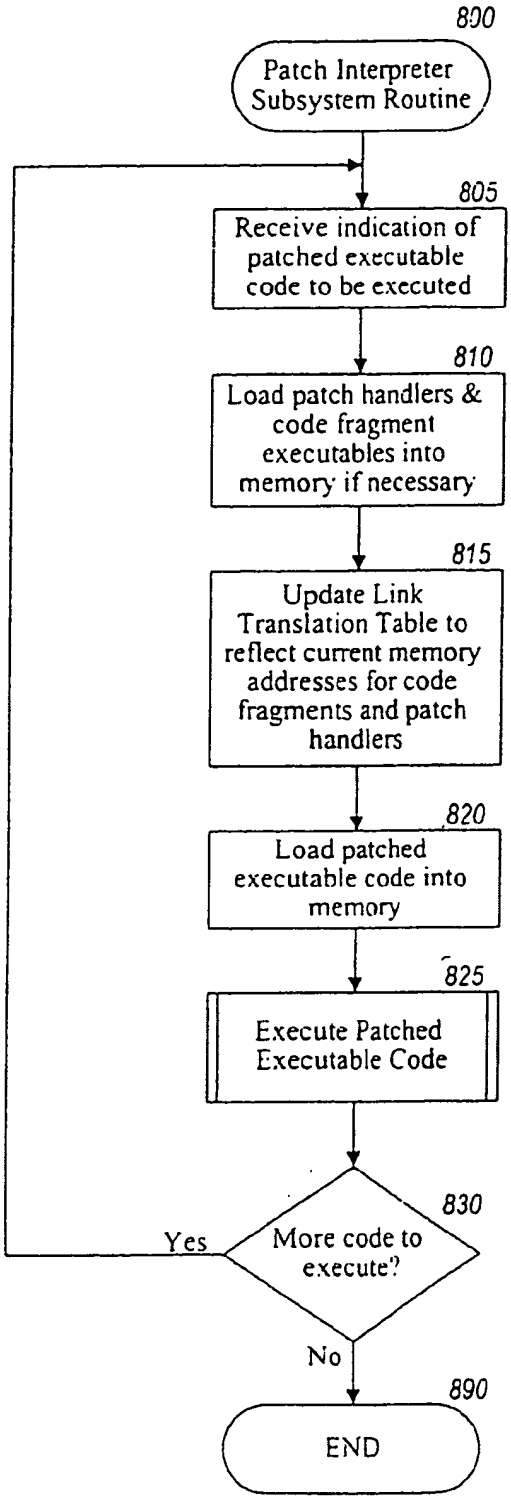

800
Patch Interpreter Subsystem Routine
805
Receive indication of patched executable code to be executed
810
Load patch handlers & code fragment executables into memory if necessary
815
Update Link Translation Table to reflect current memory addresses for code fragments and patch handlers
820
Load patched executable code into memory
825
Execute Patched Executable Code
830
More code to execute?
Yes
No
890
END

Fig. 8

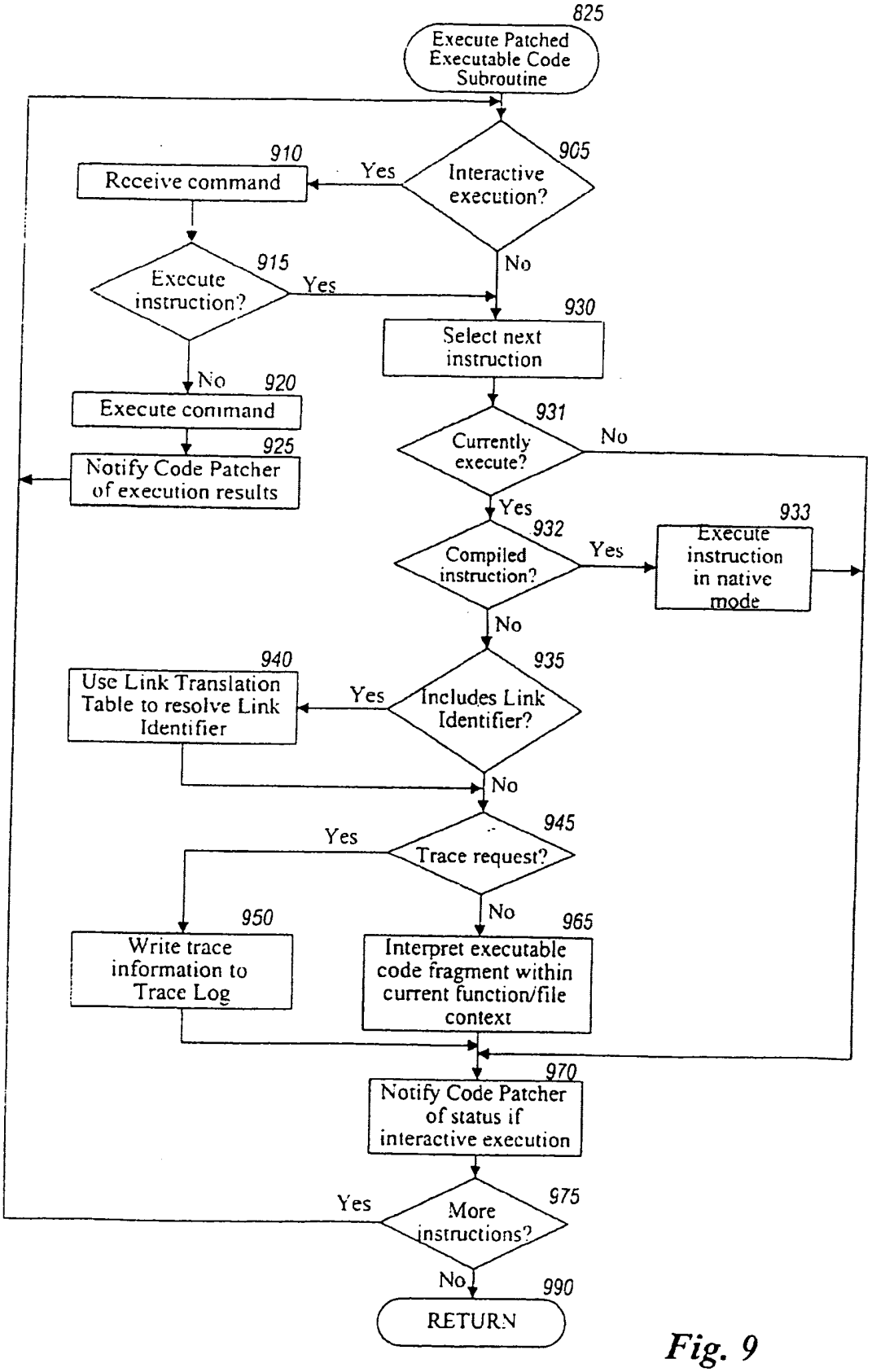

825
Execute Patched Executable Code Subroutine
905
Interactive execution?
Yes
910
Receive command
No
915
Execute instruction?
Yes
930
Select next instruction
No
920
Execute command
925
Notify Code Patcher of execution results
931
Currently execute?
No
Yes
933
Execute instruction in native mode
932
Compiled instruction?
Yes
No
940
Use Link Translation Table to resolve Link Identifier
Yes
935
Includes Link Identifier?
No
945
Trace request?
Yes
No
950
Write trace information to Trace Log
965
Interpret executable code fragment within current function/file context
970
Notify Code Patcher of status if interactive execution
975
More instructions?
Yes
No
990
RETURN

Fig. 9

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 014 263 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
18.06.2003 Bulletin 2003/25

(43) Date of publication A2:
28.06.2000 Bulletin 2000/26

(21) Application number: **99310012.2**

(22) Date of filing: **13.12.1999**

(51) Int Cl.[7]: **G06F 9/445**

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: **14.12.1998 US 212182**

(71) Applicant: **APPLIED MICROSYSTEMS CORPORATION**
Redmond, Washington 98072-9702 (US)

(72) Inventor: **Tinker, Jeffrey L.**
Kenmore, Washington 98027 (US)

(74) Representative: **Driver, Virginia Rozanne et al**
Page White & Farrer
54 Doughty Street
London WC1N 2LS (GB)

(54) Method and system for modifying executable code to add additional functionality

(57) A system for modifying a compiled executable code file by adding patches that add functionality when the modified executable code file is executed. The modifying is performed without recompiling, relinking or rewriting the code file. Adding a patch involves creating a patch handler which when executed causes the patch statements to be executed, and may involve replacing one or more existing compiled instructions in the file with patching instructions to transfer flow of execution to the appropriate patch handler. The instructions replaced by the patching instructions can also be added to the patch handler. Patches can also include code statements which form a complete module, such as an invokable routine, which can be referenced by other patch statements to cause the module to be executed. Specialized trace requests can also be added as patch statements. The trace requests will make specified information about the current execution of the executable code file available to a software developer. Patch statements can include variables and expressions that will be evaluated in the context of the appropriate current variable scope, regardless of whether the scope is defined within the patch or by existing compiled instructions. After patches have been added, they can be disabled so as to prevent their execution without removing the patching instructions from the compiled executable file. Patches can also be qualified with conditions such that the patch will be executed only when the conditions are true at the time of execution.

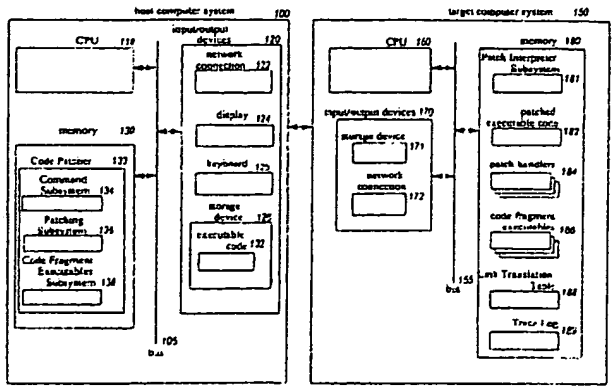


Fig. 1

EP 1 014 263 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 99 31 0012

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
X	US 5 619 698 A (EIDT ERIK L ET AL) 8 April 1997 (1997-04-08)	7	G06F9/445
A	* abstract * * column 1, line 18 - line 22 * * column 6, line 15 - line 47 * * claim 47 * ---	1-6,8-10	
A	US 5 764 992 A (TITUS DIANE ET AL) 9 June 1998 (1998-06-09) * claim 17 * ---	1-10	
A	EP 0 853 278 A (SUN MICROSYSTEMS INC) 15 July 1998 (1998-07-15) * abstract * * page 3, line 47 - line 48 * * claims 1-3,9 * ---	1-10	
A	US 5 155 847 A (PORRETT WILLIAM A ET AL) 13 October 1992 (1992-10-13) * abstract * * column 1, line 63 - line 64 * * column 2, line 10 - line 12 * * claim 1 * ---	1-10	TECHNICAL FIELDS SEARCHED (Int.Cl.7) G06F
A	US 5 694 566 A (NAGAE TAKAAKI) 2 December 1997 (1997-12-02) * abstract * * claim 1 * * column 1, line 48 - line 55 * ---	1-10	
A	US 5 699 275 A (KENNEDY III WILLIAM C ET AL) 16 December 1997 (1997-12-16) * abstract * * column 1, line 58 - line 61 * -----	1-10	

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	24 April 2003	Kusnierczak, P

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 99 31 0012

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

24-04-2003

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5619698 A	08-04-1997	NONE	
US 5764992 A	09-06-1998	NONE	
EP 0853278 A	15-07-1998	US 6253317 B1	26-06-2001
		CA 2226224 A1	09-07-1998
		EP 0853278 A2	15-07-1998
		JP 10301807 A	13-11-1998
US 5155847 A	13-10-1992	CA 1310131 A1	10-11-1992
US 5694566 A	02-12-1997	JP 6242990 A	02-09-1994
		US 6076134 A	13-06-2000
US 5699275 A	16-12-1997	AU 5440096 A	30-10-1996
		CA 2217856 A1	17-10-1996
		EP 0820614 A1	28-01-1998
		JP 11503545 T	26-03-1999
		WO 9632679 A1	17-10-1996